KU LEUVEN

FACULTEIT WETENSCHAPPEN
DEPARTEMENT NATUURKUNDE EN STERRENKUNDE
CELESTIJNENLAAN 200D
3001 LEUVEN, BELGIË

The Ultimate Binstruct Manual

A comprehensive guide

Rik Huygen • KU Leuven • Instituut voor Sterrenkunde • rik.huygen@kuleuven.be
Celestijnenlaan 200D bus 2401, 3001 Leuven, Belgium **T** +32 16 327043 **F** +32 16 327999

Issue 0.9

Document Change Record

Issue	Date	Comments
0.1	8 Sep 2010	First draft, document structure, gathering ideas, transfer text from existing package documentation, … This issue describes features available up to hcss.dp.pacs-5.0.650.
0.2	8 Nov 2010	Added section on loading a TmVersion into your HIPE session.
0.3	11 Jan 2011	Added documentation on Parameter Limits as implemented as of 6.0.1786 [HCSS-5421]
0.4	24 Jun 2011	Added documentation on extended information for parameters and ValuesExtractor.
0.5	29 May 2012	Added information to the document for HCSS Peer Review.
0.6	14 Aug 2013	Added section on archived JIRA issues
0.7	22 Oct 2013	Added information on MIB Data Formats
0.8	28 Oct 2013	Added information on PIB Data Format and on the Telemetry Versions map
0.9	30 Oct 2013	Added information about identifiers in MIB and HPSDB implementations and about the format of tm-files. Small changes in the chapter for the expert user.

Last modified by Rik Huygen on 30 Oct 2013 15:39.

Rik Huygen • KU Leuven • Instituut voor Sterrenkunde • rik.huygen@kuleuven.be
Celestijnenlaan 200D bus 2401, 3001 Leuven, Belgium **T** +32 16 327043 **F** +32 16 327999

Reference Documents

[R1] A Java Library for Describing Binary Data Structures, 2006, ADASS XV, ASP Conference Series, Vol. 351, 200-223.

[R2] SCOS-2000 Database Import ICD, S2K-MCS-ICD-0001-TOS-GIC, Issue 5.2, July 2, 2003.

[R3] HCSS MIB Clarification and Tailoring Notes, HSC/DOC/0300, Issue 1.9, September 14, 2007.

Table of Contents

Introduction 2

The User Manual 3
- Introduction 3
- A Typical User Session 3
- Packets and Parameters 5
- The Packet Sequence 5
- Identifiers 6
- Telemetry Versions 6
- Reading in your data 7

The Expert User Manual 8
- Introduction 8
- Properties 8
- Parameter Definition 10
- Packet Definition 10
- Aliases 11
- Parameter Value Extraction 12

The Developer Manual 14
- Introduction 14
- The Package Structure 14
- Relation to other packages 15
- The Design 16
- The Process 18

The Framework Developer Manual 19
- Introduction 19

The Data Guide 20
- Introduction 20
- The Telemetry Versions Map 20
- The PIB Data Format 21
- The MIB Data Format 21
- The HPSDB Data Format 22
- The tm-file Data Format 24

Appendix A: User Commands 25

Appendix B: Archived JIRA Issues 26
- HCSS-5930 • PacketReader iterator has implementation flaw 26

Glossary 27

1.Introduction

This document is a comprehensive guide of the binstruct framework for both users and developers.

The first section is the user manual. This section explains how the binstruct framework is used by general users and instrument specialists from within HIPE. The language of choice in this section is Jython.

The second section is the expert user manual. This section explain how to interact with the framework as an instrument specialist. You will learn how to switch instrument definitions and how to analyze housekeeping trends. The language of choice for this section is Jython.

The third section is the developer manual. This section explains how the binstruct framework is used by pipeline and other developers. The section focuses is on how you develop your code against the binstruct library and how to use the binstruct application programming interface (API). The language of choice for this section is Java, with some example in Jython.

The fourth section is a framework developer manual and is about developing the binstruct framework itself. This section is highly specialized and should in principle be read by anyone who is going to make changes to the binstruct framework or who is developing a service for reading instrument definitions. The language of choice for this section is Java.

Users are expected to have basic knowledge of Jython. In addition users need to understand the data structures (Product and Context) and the configuration properties used in the Herschel Common Software System (HCSS). A basic knowledge of pools, storages and tasks is certainly an advantage.

Expert users are expected to have basic knowledge of cvs.

Developers are expected to know the Java language and understand the ins and outs of the Product Access Layer (PAL) and the Configuration in the HCSS.

2.The User Manual

2.1. Introduction

Binstruct is short-hand for *Binary Structure*. It is a library to decode data that is presented in any binary form.

The binstruct package provides a framework for defining the structure of binary data in a generic way. The idea is that the structure of binary information can be defined in terms of fields. These fields can contain parameter values or can be another self-standing (sub-)structure. The package is primarily developed to help understand and decode spacecraft telemetry data, so some of the terminology used here is taken from documents like the Packet Structure Interface Control Document of ESA, which describes how telemetry and tele-command packets are structured and organized.

The definitions for these fields and structures are kept persistent in several ways. The base implementation mechanism for the definitions is via the MIB (Mission Information Base) which contains all telemetry and tele-command definitions for a spacecraft mission. Additionally another (file based) persistency mechanism is used to define structures that are not in the MIB e.g. instrument science data, diagnostic data etc.

The general user of this package is completely shielded from these persistency mechanisms and has generic methods at hand to retrieve the proper information regardless where and how it is stored.

More advanced users can decide to use different sources for the definitions of parameters and packets. Binstruct can be re-initialized dynamically in order to switch between these sources.

2.2. A Typical User Session

Let's, as a quick start guide, assume you want to see the variation on the evaporation temperature for the bolometer cryo-cooler on operational day 464. You will start reading in your data and inspecting the sequence to see what packet types it contains.

```
HIPE> seq = PacketSequence(PacketReader("/Users/rik/data/pacs/FM_FLIGHT/OD0464.tm"))
HIPE> print seq
PacketSequence class version $Revision: 1.70 $
PacketSequence contains 1230064 packets.
The 10 different packet types contained are:
      DIAGNOSTIC_HK (120 packets)
      PACS_ESSENTIAL_HK (8510 packets)
      PACS_LINK_CONNECTION (1 packets)
      PACS_NO_PRIME_HK (4372 packets)
      PACS_PHOT_HK (38178 packets)
      PACS_TC_ACP_OK (1483 packets)
      PACS_TC_EXE_COMPL (21 packets)
      PACS_TC_EXE_START (21 packets)
      PHOT_SC_BLUE (406012 packets)
      PHOT_SC_RED (169432 packets)
Number of undefined packets: 601914, for apids: [16,18,512,514,1025,1027,1158,1280,1282]
```

The packet sequence you loaded contains 1.230.064 packets of which most are science packets and undefined packets[1]. This is quite a large sequence to work with, so let's narrow down to just those packets you need to reach your goal. You know that the name of the parameter you have to inspect is BOL_TEMP_EV, so which packet types do I want to keep? The following line shows the packet types that contain values for this parameter:

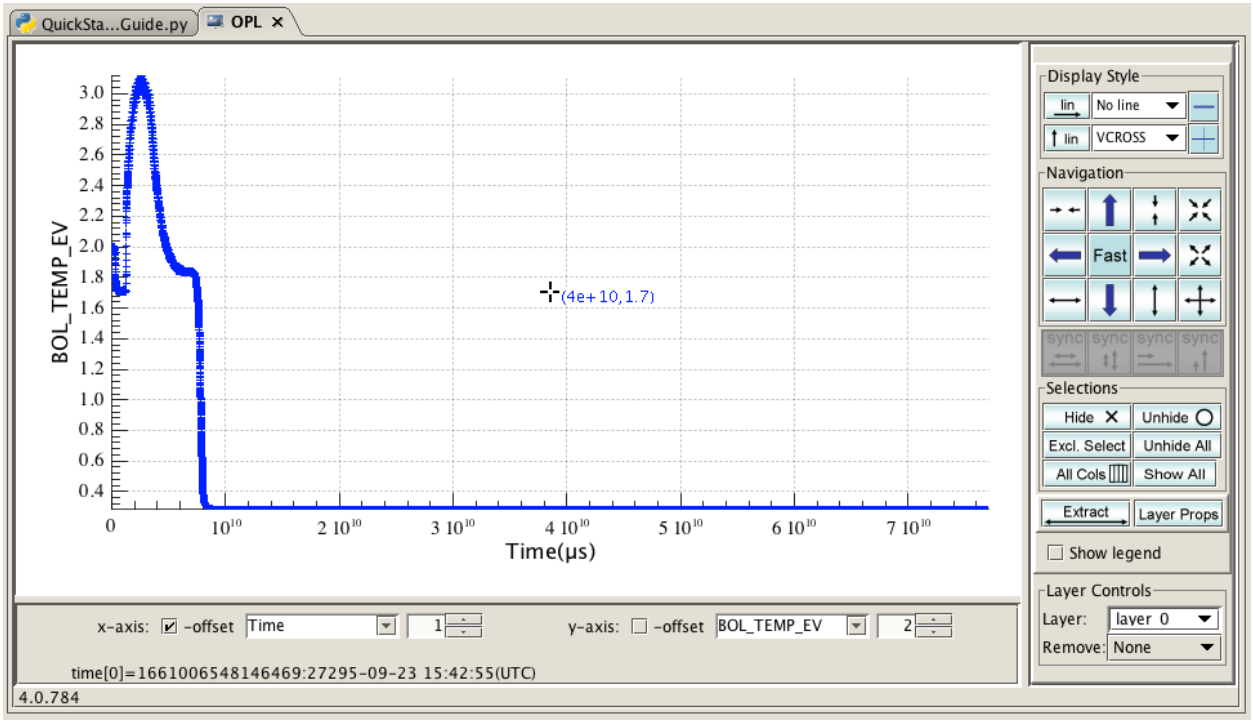
```
print seq.tmVersion.getPacketNamesFor("BOL_TEMP_EV")
[PACS_ESSENTIAL_HK, PACS_PHOT_HK, PACS_SPEC_HK, PACS_NO_PRIME_HK]
```

[1] We will explain what these undefined packets are in the expert user manual.

We obviously want the photometer housekeeping and will filter the packet sequence for these packet types into a new sequence `hk` from which we then extract the engineering values for the evaporation temperature.

```
hk = seq.select(TypeEquals("PACS_PHOT_HK"))
m = hk.getConvertedMeasures(["BOL_TEMP_EV"])
```

So here you are with the variable 'm' which contains a table with all the measures for the requested temperature sorted on time. You can quickly inspect this table with a double click on the variable m in the Variables view. This will open the TableViewer and shows the values of all the parameters with their associated time. Right click on the variable m in the Variables view to use the TablePlotter instead, which will open in the editor area as shown in the screenshot below.



If you did not know the name of the parameter you could probably find it by asking the packets which parameters they contain. Still you really want the PACS_PHOT_HK packets and from these packets you can ask which parameters it contains that match a certain string. Since you are looking for a temperature you would do something like the following:

```
hk = seq.select(TypeEquals("PACS_PHOT_HK"))
print hk[0].getParametersContained("TEMP")

array(herschel.binstruct.ParameterDefinition, [BOL_TEMP_PSU_1, BOL_TEMP_PSU_2, BOL_TEMP_R_1,
BOL_TEMP_R_2, BOL_TEMP_R_3, BOL_TEMP_R_4, BOL_TEMP_R_5, DM_SPU_LWL_TEMP, DM_CAL_SRC_TEMP,
DM_GRATING_TEMP, DM_FPU_T2_TEMP, DM_DCDC_TEMP, DM_FW_SPEC_TEMP, BOL_TEMP_FPU_ST, BOL_TEMP_EV,
DM_SPU_SWL_TEMP, DM_FW_PHOT_TEMP, DM_DSP_TEMP, BOL_TEMP_EV_SWT, DM_SPU_PS_TEMP, BOL_TEMP_FPU1,
BOL_TEMP_FPU2, BOL_TEMP_DAQ, DM_FPU_T1_TEMP, BOL_TEMP_B_2, BOL_TEMP_B_3, BOL_TEMP_B_1,
BOL_TEMP_SP, BOL_TEMP_SP_SWT, BOL_TEMP_TS, DM_CHOPPER_TEMP])
```

Since most of the packets contain several hundreds of parameters, the TEMP string filters the output to the most probable names.

If you need more information about the evaporation temperature parameter, you can ask the telemetry version for the parameter's definition:

```
pd = seq.tmVersion.getParameterDefinition("BOL_TEMP_EV")
```

Most of the definitions in this framework have a method `toStringX()` which returns extended information. The following lines show such information about the `BOL_TEMP_EV`.

```
HIPE> print pd.toStringX()
ParameterDefinition: BOL_TEMP_EV
Mib Id: PM411410
OnBoard Id: 41493
Description: BOL_TEMP_EV
Aliases: PM411410
Numerical conversion: NUMCONV_335, BOLC TEMP
```

The details of the information that is printed above is explained further down in this manual, but you can already see that several identifiers are associated with a parameter and that this particular parameter has a numerical conversion associated that allows you to immediately retrieve the temperature instead of the raw voltages.

2.3. Packets and Parameters

We will talk about packets most of the time when we speak about a binary structure. A packet is either a telemetry or a tele-command packet as it is retrieved from or send to a satellite. Packets have a fixed predefined structure which is outlined in what is called a mission information base[2] (MIB). The simplest of such structures is a packet that contains a set of parameters one after the other as in an array or list. The information base contains the definitions for all these parameters, most important, the size that a parameter takes up in the packet along with its location in the packet, but also the format of the parameter e.g. if its a floating point number or an integer.

Parameters in packets are usually in a very raw format and thus the information base describes how these raw values can be converted into more meaningful measures we call engineering values. The binstruct package provides functionality under the hood to interpret all these features while the user only has to deal with a very simple interface. Take for instance the following example which has a packet in the packet variable (for this example it doesn't matter where the packet comes from). To extract the content for a certain parameter one uses the getMeasure() method which takes the name of the parameter as an argument. The measure contains the raw value and knows how to convert this value to its engineering counterpart. The measure also has a time associated, which is derived from the creation date of the packet.

```
packet = …
m = packet.getMeasure("DPU_CPU_LOAD")
raw = m.longValue()
eng = m.getEngineeringValue()
time = m.getTime()
```

Some packets may contain more than one value for a particular parameter. In this case the getMeasure**s**() method can be used which returns an array of measures for the given parameter.

```
ms = packet.getMeasures("LCU_IV_BAND1_V")
for m in ms:
   print m.engineeringValue(),
```

2.4. The Packet Sequence

Of course there is not just one packet coming out of a spacecraft but many thousands for just one observation. The binstruct framework provides functionality to handle a huge amount of packets as a sequence. Such a packet sequence provides methods to extract raw and converted parameter values from all the packets in a sequence in one go. It also gives you the possibility to ask for a selection of packets based on their type or their content.

A packet sequence can be constructed using the default constructor and then adding individual packets using the add() or addPacket() methods. This is however a cumbersome task and not really performant at the Jython level if you need to add lots of packets. Therefore packet sequences can better be constructed from a database connection or from a binary file containing all the packets.

The following example constructs a packet sequence for a series of packets from a database. In particular, a database connection is made which will stream all packets of a certain type (1026) and for a particular observation (268437799L):

[2] Sometimes also called a satellite database, e.g. Herschel/Planck Satellite Database (HPSDB)

```
pk = PacketAccess(obsid=268437799, apid=1026)
reader = HcssConnection.getConnection(pk)
seq = PacketSequence(reader)
```

Alternatively, a packet sequence can be constructed by importing packets from a packet archive file usually called a telemetry file or shorter a tm-file[3]. The following example shows how to do this in Jython:

```
seq = PacketSequence(PacketReader("/Users/rik/data/pacs/OD423.tm"))
```

As mentioned above packets can be added to a sequence with the add() or addPacket() method, or with the fromArray() method if you have an array of packets. Please note that these methods just add packets to the end of the sequence without touching what was already in the packet sequence. Packets can be extracted from a packet sequence with the toArray() method which gives back an array of packets. Alternatively, you can address individual packets in the sequence using the Jython array syntax e.g.

```
print seq[5]
```

Packet sequence also implements the Java Iterable interface and can be used as such in Java and Jython loops. Please note this is not advisable for large sequences in Jython because the performance will drop dramatically.

To extract measures from the packet sequence there are two methods: getRawMeasures() and getConvertedMeasures(). Both methods accept an array of strings as an argument and will return a TableDataset where the first column contains the time of the measure and additional separate columns for each specified measure. Raw values are returned by the getRawMeasures() method, while the getConvertedMeasures() method returns engineering or text coded values for those parameters that have a conversion defined.

Working with full packet sequences might be not what you want because you want to address a problem with certain packet types or you might want to search for packets based on certain criteria. The select() method allows you to filter the packet sequence based on packet and/or parameter value criteria.

The following example first restricts the sequence to packets of a certain type and then extracts all the values for the given two parameters from that sequence.

```
hk = seq.select(TypeEquals("ESSENTIAL_HK"))
result = hk.getRawMeasures(["DPU_CPU_LOAD", "SPU_CPU_LOAD"])
```

Note that the above method retrieves raw values directly instead of measures. The result is a table which contains three columns, the time of each value in the first column and the raw values for the cpu loads of the DPU and SPU in the second and third column respectively.

2.5. Identifiers

In the above examples we used names to identify parameters and packets. These names are internally represented as identifiers and are case sensitive and unique. Although these names are used as keys in several *get* and *search* methods, the package does not itself enforce the uniqueness of these identifiers. This is left to the mission information base (MIB) which defines the identifiers. The name of a parameter or packet is often referred to as a mnemonic.

2.6. Telemetry Versions

What are these telemetry versions and why do I need them? How are they identified, what is the dependency with packets etc.

2.6.1. How do I get hold of a TmVersion

There are several ways to get access to the telemetry version.

[3] See chapter 6, The Data Guide, for a description of the tm-file data format.

The telemetry version registry knows about all the telemetry versions that are available. A TmVersion can be loaded from the registry with the getTmVersion() method. This method accepts a FineTime or a unique identifier for the TmVersion.

```
tmv = TmVersionRegistry.getTmVersion(time)
tmv = TmVersionRegistry.getTmVersion("FM_0.80")
```

If you don't know the identifier for a telemetry version, you can print all the versions that the registry knows about as follows:

```
HIPE> print TmVersionRegistry.toStringX()
TmVersion AVM_1.0 valid from Tue Jan 01 00:00:00 CET 1991
TmVersion AVM_1.1 valid from Mon Jul 14 00:00:00 CEST 2003
TmVersion AVM_1.2 valid from Thu Feb 19 00:00:00 CET 2004
. . .
TmVersion FM_0.81 valid from Thu Mar 05 04:00:00 CET 2009
TmVersion FM_0.80 valid from Thu Apr 16 14:30:00 CEST 2009
```

The latest version that is known can be loaded as follows:

```
tmv = TmVersionRegistry.getTmVersion(FineTime(Date()))
```

If you have access to a packet sequence or a telemetry packet, the TmVersion can be requested by the following steps:

```
seq = PacketSequence( … )
tmv = seq.tmVersion
```

This will return the telemetry version that is applicable for the packet sequence and is not necessarily the last TmVersion.

Alternatively, from a telemetry packet, the time can be extracted to pass into the TmVersionRegistry as follows:

```
tmv = TmVersionRegistry.getTmVersion(packet.timeAsFineTime)
```

2.7. Reading in your data

TODO Explain the ins and outs of PacketReader here.

3.The Expert User Manual

TODO

- Switching between definitions, when to use PIB, MIB, HPSDB. Understanding which configuration is used…
- Analyzing trend data
- Move the methods of BinstructProperties() that are listed behind the properties to a table in the developer manual as they are not of interest even to an expert user.
- Explain about aliases, parameter and packet aliases, where they are defined how and where you can use them, and how you can see which aliases are defined for a parameter or packet identifier.

3.1. Introduction

This expert guide goes in much further detail on how to work and interact with binstruct. We will explain how to configure binstruct for special purposes and how to handle special packets and parameters.

3.2. Properties

Since binstruct provides a numbers of ways to read definitions of packets and parameters, it must be configured correctly in order to find the location of the MIB ASCII files or to connect to the correct database or local store. This section describes all the properties that can be used to configure binstruct.

As a framework, binstruct is used by different instrument, and these instruments use binstruct in different ways and therefore configure it differently. For that purpose we have build in a mechanism that allows instrument specific properties for each of the default properties below. Binstruct determines the instrument name from a system wide property `var.hcss.instrument`, which contains the name of the instrument currently used in data processing.

The algorithm that binstruct uses to select and use an instrument specific property is the following:

1. Binstruct reads the value of the property `var.hcss.instrument`.
2. If the instrument is specified in the above property, binstruct constructs an instrument specific property by inserting the instrument name in lower case after the `hcss.binstruct.` part of the property. For example, for the property `hcss.binstruct.services`, an instrument specific version for PACS is constructed as `hcss.binstruct.`**`pacs`**`.services`.
3. Binstruct reads the instrument specific property and uses its value if it is set.
4. If the instrument specific property does not exist or is not set, binstruct will use the value of the default/generic property, i.e. the one without the instrument name in it.
5. For any request for the value of a property, binstruct returns the string *unknown* if neither the instrument specific property, nor the default is defined.

The properties below are the default properties that are used when no instrument specific properties are set. For the name of the property for your instrument, insert the instrument name in lower case. Please note that the deprecated properties are still interpreted as a fall back solution, but there is now instrument specific version of these properties.

Also, some of the property defaults are defined in terms of the `var.hcss.dir` property, so make sure this is properly set to your installation directory.

`hcss.binstruct.factory` • **`[BinstructProperties().getFactoryName()]`**

the class name of the factory to be used to create binstruct specific objects. This factory should be extended from AbstractBinstructFactory instead of implementing the BinstructFactory directly. This makes the factory easily extendable. The default is set to `herschel.binstruct.DefaultBinstructFactory`.

`hcss.binstruct.services` • **`[BinstructProperties().getServicesName()]`**

the class name of the interface to be used for reading the telemetry definitions from different input sources. The default is set to `herschel.binstruct.pib.PibServices`.

`hcss.binstruct.registry.tmversion.loader` • **`[BinstructProperties().getTmVersionRegistryLoaderName()]`**

The name of the implementing class for the TmVersionRegistryLoader interface. There are a few loader implementations available in the framework, see the section on writing your own telemetry version registry loader. The default for this property is set to `herschel.binstruct.TmVersionRegistryPibLoaderAscii`.

`hcss.binstruct.mib`

deprecated, use `hcss.binstruct.mib.ascii.location`

`hcss.binstruct.mib.source` • **`[BinstructProperties().getMibSource()]`**

the nature of the MIB source, i.e. pal or ascii. The default is set to ascii.

`hcss.binstruct.mib.ascii.location` • **`[BinstructProperties().getMibLocation()]`**

the location of the MIB ASCII data. This points to the directory that contains all the different versions of the MIB. This directory usually contains a number of versioned sub-directories and a number of TmVersions*.tbl files

`hcss.binstruct.mib.ascii.tm_version_map`

the name of the TmVersions table that contains information on the MIB versions and their validity periods.

`hcss.binstruct.mib.pal.store.id`

the identifier of the storage as defined by the StorageManager

`hcss.binstruct.mib.pal.poolname`

renamed to `hcss.binstruct.mib.pal.pool.name`

`hcss.binstruct.mib.pal.pool.name`

the name of the pool that contains the MIB products, this property will be used only if the `hcss.binstruct.mib.pal.store.id` is not defined.

`hcss.binstruct.mib.pal.pool.type`

the type of the pool that contains the MIB products, this property will be used only if the `hcss.binstruct.mib.pal.store.id` is not defined. If the property is not set, a local store will be used by default.

`hcss.binstruct.mib.pal.tm_version_map`

the mapId of the telemetry version map in the storage

`hcss.binstruct.mib.pal.database` • **`[BinstructProperties().getMibDatabase()]`**

the name of the database that stores the telemetry versions. This property is used by specific factory classes that create a product storage from a DbPool or HttpClientPool.

Used by the MIB PAL Services to create an instance of the DbPool that contains the MIB definitions.

`hcss.binstruct.ip_filename`

deprecated, use `hcss.binstruct.pib.ip_filename`, because this property is specific for the PibServices.

`hcss.binstruct.pib.ip_filename` • **`[BinstructProperties().getPibIpFilename()]`**

the name of the file that contains a list of all the definitions that are available in the PIB. This file is usually called instr_props.ip.

hcss.binstruct.tm_version_map

deprecated, use `hcss.binstruct.pib.tm_version_map` or `hcss.binstruct.mib.ascii.tm_version_map`

hcss.binstruct.pib.tm_version_map • `[BinstructProperties().getPibTmVersionMap()]`

the name of the TmVersions table that contains information on the PIB versions and their validity periods.

hcss.binstruct.pib

deprecated, use `hcss.binstruct.pib.location`

hcss.binstruct.pib.location • `[BinstructProperties().getPibLocation()]`

the location of the PIB data. This points to the directory that contains all the different versions of the PIB. This directory usually contains a number of versioned sub-directories and a number of TmVersions*.tbl files.

hcss.binstruct.pib.format • `[BinstructProperties().getPibFormat()]`

the format of the PIB data. Since HCSS Release 7, the PIB is available and distributed as a jar-file in order to reduce the number of files and therefore the size of the distribution. The value for this property is set to jar-file.

3.3. Parameter Definition

The basic building blocks in binstruct are packets and parameters. This section focuses on parameters, how they are defined, how they are connected to packets and what information you can get out of them.

3.3.1.

3.3.2. Parameter Limits

Another property of parameter definitions is the limits that can be associated with them. A parameter can hold zero or more limits which can be of different type and purpose. There are two classes that are needed to fully define parameter limits. The `ParameterLimitDefinition` defines the properties that are common to all the limits associated to the same parameter, e.g. the type of the limit or if the limit is to be interpreted in raw or engineering values. The parameter limits themselves are defined in the class `ParameterLimit`, which contains the lower and higher limit values for the parameter and provides a convenience method to test if a value is within limits. The next section describes how to get hold of these limits and make use of them in your code.

Using limits in your code

The following example code shows how to retrieve the limit information for a particular parameter.

```
tmv = TmVersionRegistry.getTmVersion("FM_0.105")
pd = tmv.getParameterDefinition("")
pld = pd.getParameterLimitDefinition()
pls = pd.getLimits()
```

If you don't know which telemetry version to use, you can get a list of available telemetry version by printing extended information from the registry (see also the section on getting hold of telemetry versions).

```
print TmVersionRegistry.toStringX()
```

3.4. Packet Definition

TWB

3.4.1.Undefined Packets

In some cases, a telemetry version is used which does not contain all the definitions of packets that are in the observation. This can happen when e.g. an instrument database is used while reading a complete operational day into your session. When printing information about the packet sequence, you will see that there are quite some undefined packets.

```
HIPE> print seq
      …
      PHOT_SC_BLUE (406012 packets)
      PHOT_SC_RED (169432 packets)
Number of undefined packets: 601914, for apids: [16,18,512,514,1025,1027,1158,1280,1282]
```

The solution is to use the correct telemetry version that contains all the definitions for all the packets and parameters that are downlinked by the satellite. This is usually the last available satellite database (HPSDB).

Alternatively, undefined packets can be removed from a packet sequence as follows.

```
from herschel.binstruct.util import Not
seq = seq.select(Not(TypeUndefined()))
```

3.5. Aliases

Binstruct makes extensive use of identifiers for both parameters and packets. These identifiers serve two purposes (1) they uniquely identify a parameter or packet, and (2) they provide a way of naming the parameters and packets. It is the latter that will be explained in this section.

Each parameter and packet in binstruct has a unique identifier and a number of (also unique) aliases. The identifier of a parameter or packet might be cryptic and difficult to remember, but it is possible to use more readable aliases to identify parameters and packets.

Defining aliases is done in a special MIB table and is explained in the Data Guide.

3.5.1. Parameter Aliases

In principle, aliases for parameter identifiers can be used wherever the main identifier can be used. In practice, that is in the following methods of PacketSequence:

```
m = seq.getRawMeasures(["AN_IDENTIFIER", "AN_ALIAS"])
m = seq.getConvertedMeasures(["AN_IDENTIFIER", "AN_ALIAS"])
```

and in the following methods of individual packets, e.g. HkBinStruct:

```
m = seq[0].getMeasure("AN_ALIAS")
m = seq[0].getMeasures("AN_ALIAS")
```

For the ValuesExtractor utility, parameter name aliases are supported for the getRawValues and getEngineeringValues methods, not for the getDiagnosticValues.

```
table = ValuesExtractor.getRawValues(seq, "AN_ALIAS")
table = ValuesExtractor.getEngineeringValues(seq, "AN_ALIAS")
table = ValuesExtractor.getDiagnosticValues(seq, "NO_ALIAS_SUPPORT")
```

There are several ways to ask for the parameters that are defined in binstruct, depending whether you want to see all parameters defined, only those parameters that are contained in a certain packet, or even a sub-set of those.

To see which parameters are contained in a packet, the easiest way is to ask the packet itself. Suppose you want to see the parameters from the PACS_PHOT_HK packet which you know is in your packet sequence, first filter the sequence for the packet type, then ask a packet for its parameters.

```
hk = seq.select(TypeEquals("PACS_PHOT_HK"))
print hk[0].getParametersContained()
```

This will produce a long list of parameter names, currently this packet holds about 700 definitions. To refine the search, you can add a pattern as an argument. The following example print only those parameters that have the word TEMP in their identifier.

```
print hk[0].getParametersContained("TEMP")
```

If you now want to learn if one of these parameters has aliases, request the parameter definition from the packet and check if it has aliases:

```
pd = hk.tmVersion.getParameterDefinition("BOL_TEMP_EV")
HIPE> print pd.hasAliases()
True
HIPE> print pd.getAliases()
[PM411410]
```

As you see, the alias is quite cryptic because this is the MIB identifier (mibId). The instruments do not use the mibId as an identifier because it is not suitable as an easy to remember mnemonic. Instead the description of the parameter is used as the primary identifier and the mibId is set as an alias by default (see The Data Guide chapter for more information on which identifiers are used depending on the format of the telemetry version that is used).

3.5.2. Packet Aliases

3.6. Parameter Value Extraction

3.6.1. From a single packet

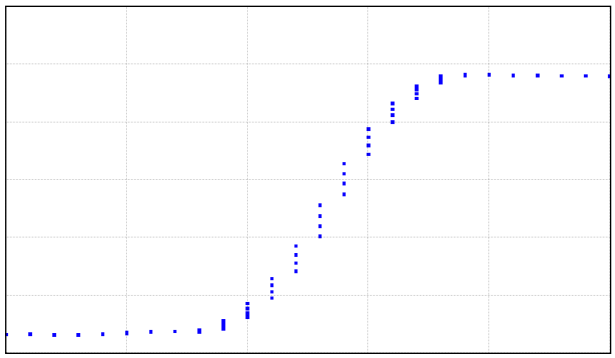
TBW

3.6.2. From a packet sequence

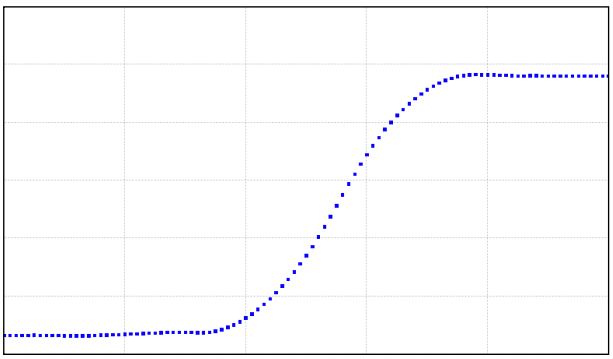
The recommended way to get parameter values out of a packet sequence is to use the ValuesExtractor utility class. Especially when you are working with special parameters and packets that possibly contain more measures for the same parameter in one packet. The getRawMeasures() and getConvertedMeasures() of PacketSequence discard all values with the same time information and remember only the last extracted value. ValuesExtractor doesn't do anything with your data points, they are not time sorted and they are not removed in the case of equal time. ValuesExtractor extracts each parameter value from each packet in the order they appear in the packet and in the sequence.

```
seq = PacketSequence(PacketReader("/Users/rik/data/pacs/test-data/OD0736.tm"))
tm = seq.select(TypeEquals("514_3_25_10500_0"))

v = ValuesExtractor.getEngineeringValues(tm, "AESA6001")

l = Long1d([0, 25, 50, 75])*10000L
l = RESHAPE(REPEAT(l, len(v["Time"].data)/4))
v["fixedTime"] = Column(v["Time"].data+l)
openVariable("v", "Dataset Viewer")
```

The above code shows the use of the ValuesExtractor for a packet that contains values from 4Hz sampling of a parameter named AESA6001. The time that is associated with each value is the packet creation time. Consequently, in the dataset table `v`, there will be each second a set of four values with the same time. The three lines after the ValuesExtractor call adds another column to the dataset where the times are fixed for 4Hz sampling. It makes a considerable difference in your data processing.



The above figure shown a plot of `v` with the original uncorrected time information, for each time you have four values. The figure below show the same zoomed plot using the corrected time information.



4.The Developer Manual

TODO

- Describe the differences between MIB and HPSDB implementations, even ASCII and PAL. E.g. formatVersion of the TmVersions table (ASCII) versus TmVersion Map in PAL.
- How should a developer read telemetry and manipulate its content. Explain how this is different from what the user normally does.
- How can a developer optimize his/her code for reading large amounts of data.

4.1. Introduction

The herschel.binstruct package provides a framework for using and working with binary data represented in a native Java byte array, i.e. byte[]. The byte array is used as a generic structure that contains all kind of data as data fields. These data fields do not need to be byte aligned in the array, but can start at any bit location and be of any size and type as long as it fits into a primitive Java data type. The work-horse for accessing these fields in a byte array is a class called ByteArrayManipulator which has a number of static methods to get data out and put data into the byte array.

Data fields can be Java primitive data types, but can also be defined as parameter values called Measures. These measures are then associated with their parameter definitions and can be raw values or converted values depending on the available conversion. Converted values are also called engineering or calibrated values. Parameter definitions and conversions are kept in a definitions database. The package provides access to different formats of this database.

The underlying framework keeps track of all definitions and how they are associated with binary structures and parameter values.

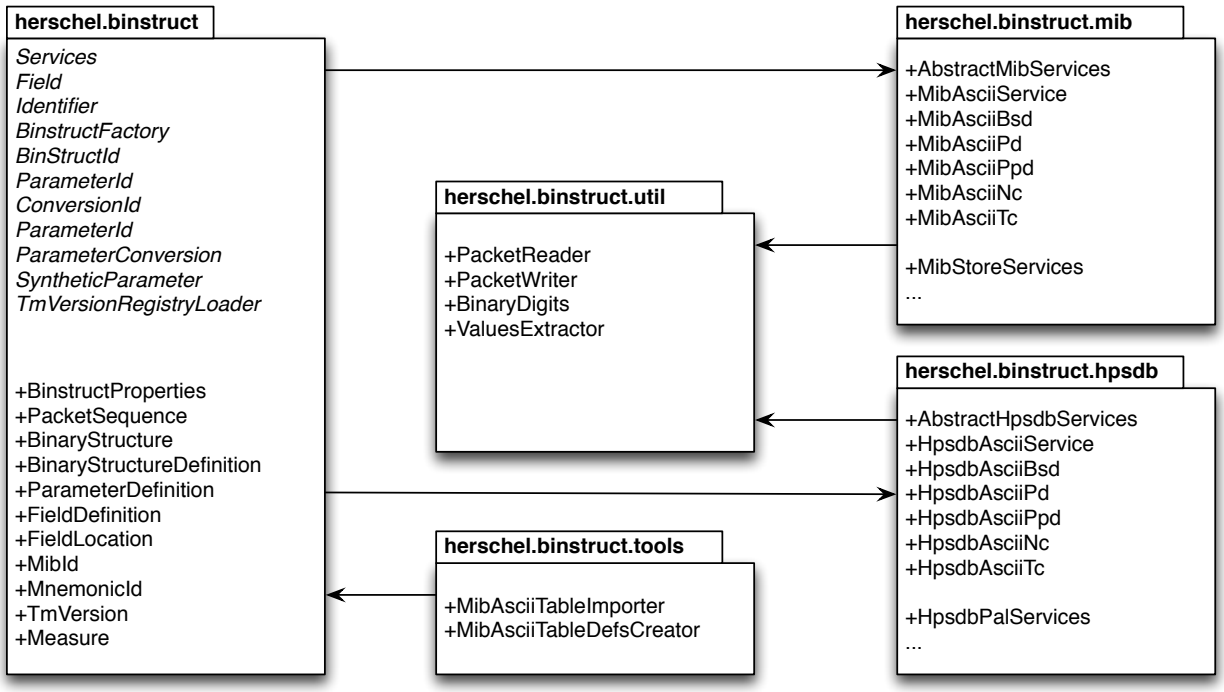
4.2. The Package Structure

The herschel.binstruct package contains the main classes and interfaces that provide all the functionality for BinaryStructure and its related classes like ParameterDefinition and Measure. The package contains four sub-packages, which contain more or less decoupled classes or completely independent utility classes.

Access to the different formats of the databases is separated out into sub-packages like herschel.binstruct.mib and herschel.binstruct.hpsdb. These sub-packages contain implementations of the Services interface for that specific database format. The MIB stands for Mission Information Base and is a ASCII file based relational database. Each file contains a table in the relational model. The HPSDB is the Herschel Planck Satellite DataBase is almost identical to the MIB format, but contains some satellite specific extensions. Another format which is not shown in the diagram below (because it will be phased out) is the PIB, a Persistent Information Base which is a ASCII file based object database. Each file represents a persistent object in the object model.

The herschel.binstruct.util sub-package originally contained a number of general utility classes on which binstruct heavily relies, e.g. the core extraction of data fields from the byte[] that is done in the ByteArrayManipulator class, and the Predicate interface that provides a means of evaluating conditions used to select packets and parameters based on certain criteria. Both the ByteArrayManipulator and the Predicate classes have been moved out of herschel.binstruct into the herschel.share package. The util sub-package now only contains classes that read in and write out telemetry data, manipulate sorted tables and a few specific implementations of the Predicate interface.

TODO: SHould the implementation of the Predicate interface also move to herschel.share?



The herschel.binstruct.tools sub-package contains application classes to support the administration of binstruct. It contains currently a class to create the definition files for importing ASCII tables into the Herschel specific TableDataset, and has a tool to compare different telemetry versions.

TODO: The Reporter class and the MibAsciiTableImporter class should move into the util sub-package.

4.3. Relation to other packages

Although we tried to develop the binstruct package as an independent self-standing package, it still contains a number of dependencies to other HCSS packages that are inevitable. This section provides insight in these dependencies and either describes the reason for the dependency or gives a possible route for refactoring to get rid of the dependency.

`herschel.ccm.api`

The most obvious dependency to this package is to the TmSourcePacket class. Binstruct was mainly developed to extract and interpret the telemetry data from the Herschel satellite. Other dependencies to this ccm package are all related to the implementations of the Services interface.

`herschel.ia.dataset`

The files from the mib and the hpsdb packages are all loaded into a TableDataset which is only used internally and as an intermediate product. The PacketSequence class has methods that return a TableDataset that contains the values of parameters that are contained in the sequence. Other datasets are not used by binstruct.

`herschel.ia.io`

The dependency to this package is completely due to reading in the ASCII tables for the MIB.

`herschel.ia.pal`

Since telemetry definitions can be loaded from a Product storage, there is obviously a dependency to the PAL. The dependency is kept to a minimum and limited to the factory classes and the registry loaders and the mib and hpsdb sub-packages.

`herschel.ia.numeric`

binstruct is quite strongly coupled to the numeric package. This is mainly for two reasons (1) the numeric types are used in the columns of the TableDataset objects that are used in binstruct, and (2) for the numerical conversion that return engineering values for parameters.

herschel.mib

There is only one dependency between binstruct and the mib package, that is to calculate the width of a parameter value based on its PTC and PFC values. This dependency is one of the candidates for refactoring.

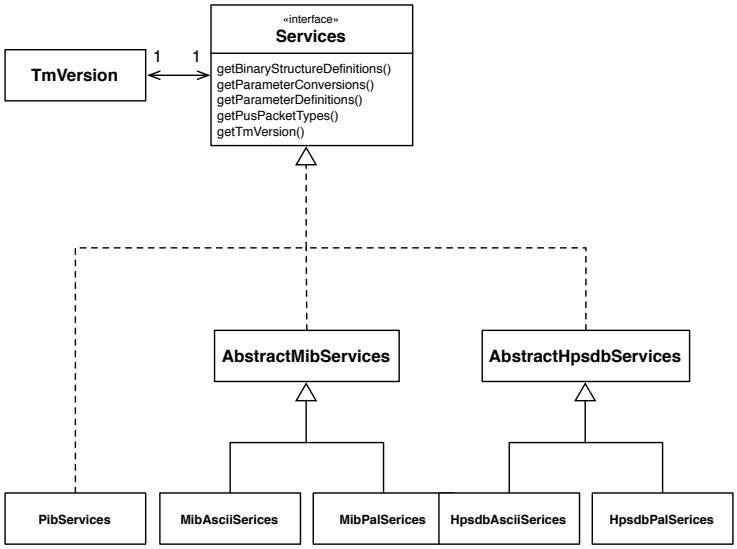
herschel.share

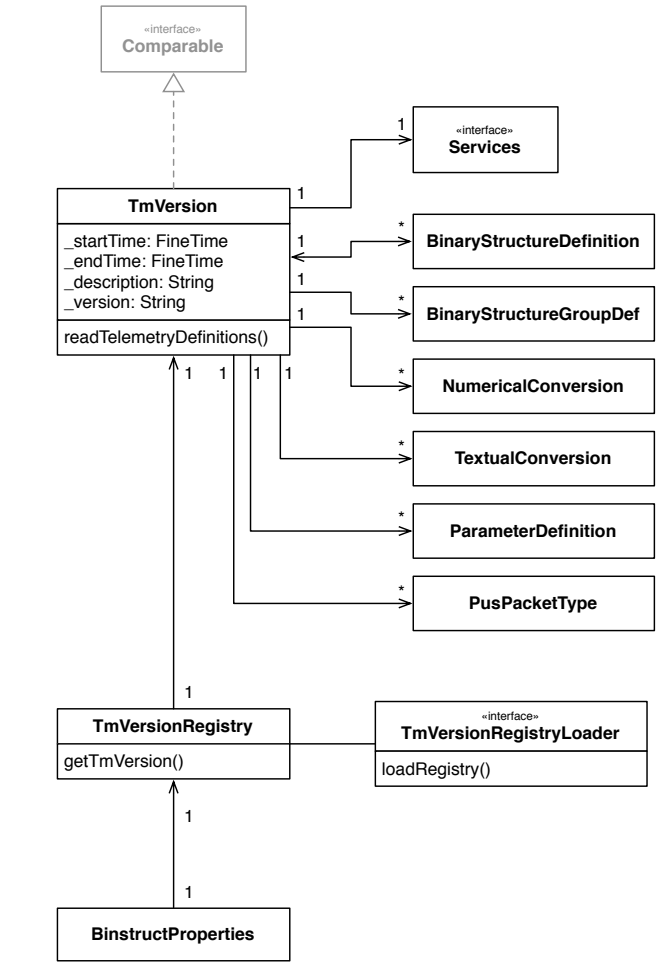
herschel.share is by definition a package with generic classes that are shared. Binstruct makes use of the time conversion functions in fltdyn and uses a few classes from the unit package. The Configuration class is used by the BinstructProperties and by the applications in the binstruct.tools sub-package. There are a number of specific binstruct implementations of the Predicate interface like TypeEquals or MeasureGreaterThan. From the util sub-package, the SizedIterator interface is implemented in a number of anonymous classes that provide the iterators for the Services interface, and the ByteArrayManipulator class was originally a binstruct utility class which moved here so other packages could make use of this without adding a dependency to herschel.binstruct. Also the CRC16 class is used to calculate the checksum of a telemetry source packet.

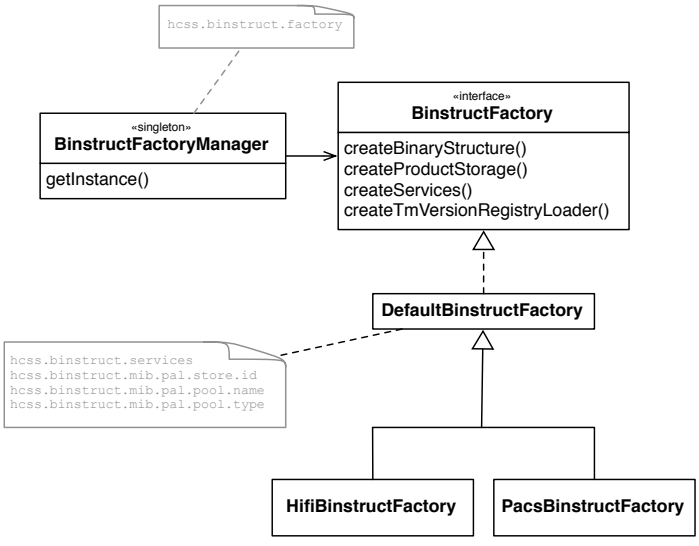
4.4. The Design

This section will describe the complete design of the binstruct framework.

4.4.1. The inner core classes



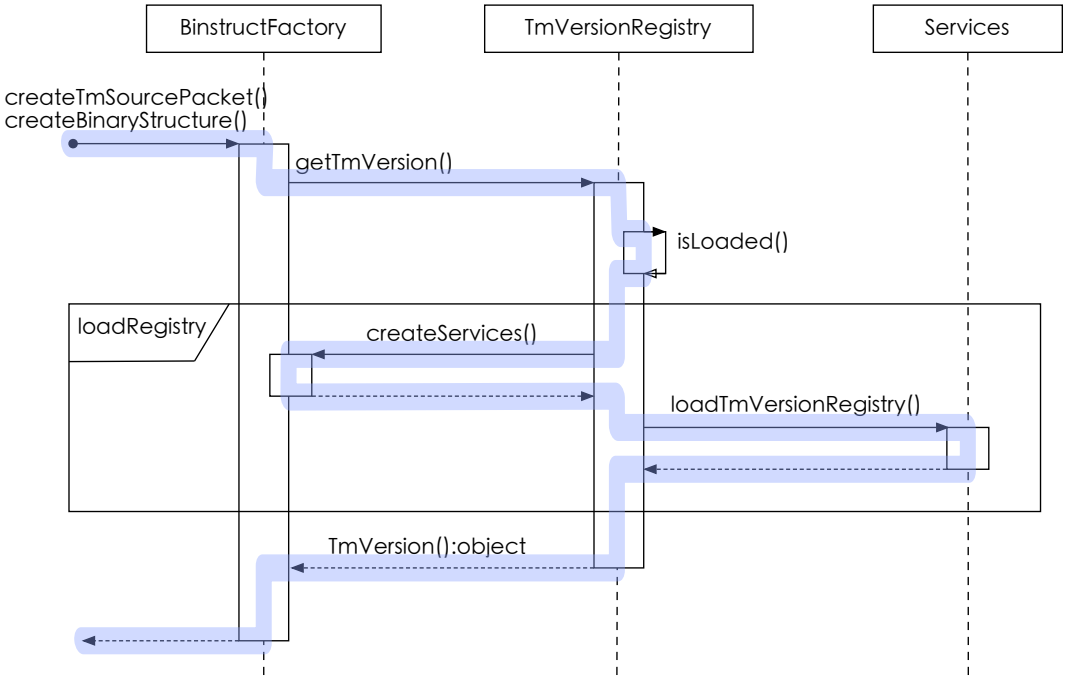

«interface»
Comparable
«interface»
Services
TmVersion
_startTime: FineTime
_endTime: FineTime
_description: String
_version: String
readTelemetryDefinitions()
BinaryStructureDefinition
BinaryStructureGroupDef
NumericalConversion
TextualConversion
ParameterDefinition
PusPacketType
TmVersionRegistry
getTmVersion()
«interface»
TmVersionRegistryLoader
loadRegistry()
BinstructProperties


hcss.binstruct.factory
«singleton»
BinstructFactoryManager
getInstance()
«interface»
BinstructFactory
createBinaryStructure()
createProductStorage()
createServices()
createTmVersionRegistryLoader()
DefaultBinstructFactory
hcss.binstruct.services
hcss.binstruct.mib.pal.store.id
hcss.binstruct.mib.pal.pool.name
hcss.binstruct.mib.pal.pool.type
HifiBinstructFactory
PacsBinstructFactory

4.5. The Process

4.5.1. Creating a BinaryStructure Object

There are several ways to create a BinaryStructure object. We can categorize them into methods for the normal development use and specialized forms for specific cases. This distinction is made because a BinaryStructure can have an associated definition that describes its type, size and the format of parameters. These definitions are loaded from objects that are connected to a specific InstrumentConfiguration based on the instrument model and the time. Its is rather complicated and cumbersome to find and associate the correct definitions to a binary structure. We therefore provide a factory method that does all this work for you. The factory is called BinstructFactory.



5.The Framework Developer Manual

TODO

- developing your own service, example case…
- explain the design and the dependencies between TmVersions, ParameterDefinitions, PacketDefinitions, what are
- explain how to implement the TmversionRegistryLoader interface, e.g. for HIFI to load their TmVersions from the ia_obs_cal

5.1. Introduction

TBW

6.The Data Guide

TODO

- Describe in detail which fields are used from which MIB/HPSDB tables and for what purpose.
- Describe the tailoring document and what its purpose was and what the most important changes are.
- Describe the differences between the Tailored MIB and the HPSDB.

6.1. Introduction

This section explain data formats used by binstruct.

6.2. The Telemetry Versions Map

The telemetry versions map is a table that describes the different versions of the PIB/MIB/HPSDB definitions. In the course of the mission, changes will be implemented in the definitions based on progress, new components, improvements or new insights, and this will introduce new versions of the telemetry definitions database. Most of the time, new definitions are just added and existing definitions will stay untouched, so the new telemetry version will then replace the previous version. Sometimes however, a new version overwrites a previous version because it changes an existing definition and makes it incompatible with the previous version. That means the definition is only valid as of a certain time while the older definition is valid only before the change. A time dependency is introduced between the telemetry versions.

Telemetry versions can exist in different formats and the book-keeping of the versions is done in a registry. The characteristics of the telemetry version are loaded into the registry using the specific TmVersionRegistryLoader for that format.

In the ASCII format of the PIB, MIB and HPSDB, the telemetry versions map (TmVersions.tbl) is an ASCII file in csv[4] format and is loaded into the registry by TmVersionRegistryPibLoaderAscii or TmVersionRegistryMibLoaderAscii. The file is by default located in the parent directory relative to the database tables. Both the location and the name of this file can be changed by a property, see the section on Properties in chapter 3 ‘The Expert User Manual’. The file contains a list of telemetry versions, one per line, with their associated start and end time that defines the validity period for that telemetry version. The end time field can be empty in which case the validity runs until the start of the next telemetry version, or until the current time. The version identifier (field 2) is used within binstruct to identify and select the telemetry version, the location field (field 5) contains the name of the directory where this version is located relative to the location of the TmVersions.tbl file. The location name is often the same as the version identifier.

```
# File
#   TmVersions.tbl
#
# Description
#   Each line in this file defines a version of the MIB with its validity period.
#
# Format
#   Each line is in comma separated value (CSV) format with ';' as delimiter
#   Field 1: The keyword 'TmVersion'
#   Field 2: Version Identifier
#   Field 3: Start time (in UTC)
#   Field 4: End Time (in UTC, can be empty)
#   Field 5: (only for fileFormatVersion >= 3.0) Location for this TmVersion
#   Field 6: Description
```

[4] csv = comma separated values

```
FileFormatVersion;3.0

TmVersion;FM_0.104;2009-01-14T21:00:00Z;;FM_0.104;New Configuration, PACS update of DPU OBSW on nominal side
TmVersion;FM_0.105;2009-01-14T21:00:01Z;;FM_0.105;New Configuration, PACS update of DPU OBSW on redundant side
TmVersion;FM_0.106;2009-02-25T03:00:00Z;;FM_0.104;Identical to FM_0.104, SFT HeI at Kourou, new DPU and SPU OBSW
TmVersion;FM_0.107;2009-03-05T03:00:00Z;;FM_0.105;Identical to FM_0.105, SFT HeI at Kourou, new DPU and SPU OBSW
TmVersion;FM_0.108;2009-05-24T11:00:00Z;;FM_0.104;Identical to FM_0.104, SFT HeII from ESOC, after launch
```

Note that the file format is currently version 3.0. Format version 2.0 introduced start and end time in FineTime instead of just a Date string, and format version 3.0 introduced the location of the telemetry version to allow a different naming for the directory.

In the PAL format of the MIB and HPSDB (there is no PAL format for the PIB), the telemetry versions map is a MapContext of type “TmVersions”. This TmVersions map contains known MIB/HPSDB products in the database. The TmVersionsRegistry is build from the meta data of each such product that is associated to the TmVersions MapContext. This is done in the TmVersionRegistryPalLoader. The format version of the TmVersions map is determined by the mandatory formatVersion meta data of the MapContext. While format versions 1.0 and 2.0 were implemented as a TableDataset containing the start/end time, identifier, and urn[5] of each MIB/HPSDB product, format version 3.0 is based on a MapContext and the products are directly associated to this context.

The TmVersions contexts are identified in the PAL with a tag which is defined by the property `hcss.binstruct.mib.pal.tm_version_map` also known as the mapId. This allows for several different TmVersions maps to co-exist within the database, e.g. a different map for each instrument or for different purposes.

6.3. The PIB Data Format

This format is historical and will be phased out. Instead of laying out the definitions in a relational database format, i.e. tables as done in the MIB/HPSDB, in the PIB the properties of each object, e.g. ParameterDefinition or BinaryStructureDefinition are written out to a file with the name of the file identical to the unique mnemonic identifier of the object. The idea was to have a simple object oriented approach, but this quickly resulted in directories containing more than 60.000 files for each telemetry version. Because of problems with the versioning and the build system, it was decided to pack the files into a zip archive (a jar-file) for each telemetry version, which can be read by binstruct without unpacking. Both the individual files and the zip archive can still be used with binstruct, but this format is no longer maintained.

6.4. The MIB Data Format

The complete MIB format is described in the SCOS-2000 Database Import ICD [see R2]. This document describes all the parameter and packet definitions, calibration parameters, synthetic parameters, conversions etc. Not all these definitions are fully supported by binstruct. In this section we will describe what is supported by binstruct and how it is used.

Definitions are described in tables to be read by a relational database. These tables are in ASCII format which are well described in the SCOS-2000 Database Import ICD [see R2]. The following tables are read by binstruct:

PCF Parameter Characteristics File containing the definitions of the monitoring parameters. The information from this table is mainly used for construction ParameterDefinition objects.

PLF Parameter Location File, defining the location of the parameters in the fixed TM packets. The information in this table is used to construct field locations of parameters in packets (BinaryStructureDefinition).

PID Packet Identification File, containing the definition of TM packets and their correspondence with the packet identification fields (e.g. APID/type/subtype). This information is used by BinaryStructureDefinition objects and PusPacketType objects.

[5] urn = Uniform Resource Name, used within the HCSS to uniquely define the location of products in the PAL.

PIC Packet Identification Criteria File, containing the definition and position of the additional identification fields for each packet type/subtype combination. This additional identification information is used by the PusPacketType objects.

VPD Variable Packet Definition File, detailing the contents of variable TM packets. The information in this table is used by the BinaryStrucureDefinition.

OCF OutOfLimits Checks File, defining the characteristics of all the checks applied to a specified monitoring parameter. Limits are a property of ParameterDefinition.

OCP OutOfLimits Definition File, defining the allowed (ranges of) values for monitoring parameters.

PMA Parameter Mnemonic Aliases, defining the aliases for parameter mnemonics. These aliases can be used in selection criteria etc. and are associated to the ParameterDefinition object. This table is binstruct specific and is not part of the MIB delivery. The table is optional.

PCA PacketMnemonic Aliases, defining the aliases for packet mnemonics. These aliases can be used in selection criteria etc. and are a associated to the packet definition object BinaryStructureDefinition. This table is binstruct specific and is not part of the MIB delivery. The table is optional.

OBW On-Board Width defines the parameter width for on-board processes and is not related to the padded width of a parameter as specified in the PCF. This table is binstruct specific and is not part of the MIB delivery. The table is optional.

CAF Calibration Curve File, defining the numerical calibration curves. This and the CAP table are used to construct NumericalConversion objects associated to ParameterDefinitions.

CAP Calibration Curve Definition File, defining all the raw/engineering value couples for each numerical calibration curves

TXF Text Strings Calibration Curve File, defining the textual calibration curves. This and the TXP table are used to construct TextualConversion objects associated to ParameterDefinitions.

TXP Text Strings Calibration Curve Definition File, defining all the raw/string value couples for each textual calibration curves

To make things more complicated, the instruments have made a few changes to the official MIB format during their development phase. These changes have been collected in the HCSS MIB Clarification and Tailoring Note [see R3].

6.4.1. Identifiers

For the MIB ASCII and PAL implementations, the identifiers for parameters and packets are derived directly from the PCF_DESCR and PID_DESCR fields respectively. The Tailoring document [R3] states explicitly that these two fields must be unique per instrument, and their context can thus be used as a unique identifier for parameters and packets.

6.5. The HPSDB Data Format

The Herschel Planck Satellite Database is essentially a combined MIB for the complete satellite system. That means the individual MIBs from the different instruments and the MIB from the spacecraft are merged into one MIB, called the HPSDB. Unfortunately, in the merging process, industry was allowed to make a few changes to the MIB format (a bit like the tailoring of the original MIB by the instruments).

This section will focus on the changes between MIB and HPSDB that are relevant to the binstruct package and describe those in detail. We will not again describe the MIB format as this was already extensively done in the previous section.

Basically, the changes are the following:

- Two additional tables are used by the HPSDB services.
- The naming of the monitoring parameters has been changed, see the section on Identifiers below.

- The naming of the packets has been changed, see the section on Identifiers below.

MCF Polynomial Calibration Curve Definitions, defining the coefficients of the polynomial function used for calibration. This table is used to construct NumericalConversion objects with a polynomial function, associated to ParameterDefinitions.

SYN Synthetic Parameter List

6.5.1. Identifiers

For the HPSDB identifiers are more difficult to define since this is essentially a merged MIB and the PCF_DESCR and PID_DESCR fields only need to be unique per instrument. Therefore another pragmatic method is used to define the unique identifiers for parameters and packets. We checked that there are no duplicate descriptions for parameters between the instrument MIBs and we gave the choice to the instruments to decide.

For the HPSDB ASCII implementation, parameter identifiers are derived from the PCF_DESCR field if the parameter is an instrument parameter[6], otherwise the identifier is derived from the PCF_NAME field which contains the cryptic database identifiers that make it much harder to remember.

For the HPSDB PAL implementation, only for HIFI instrument parameters the identifier is derived from the PCF_DESCR field, the identifiers for the PACS and SPIRE instrument parameters and the satellite parameters are derived from the PCF_NAME field.

Things get a little more complicated for packet identifiers.

For the HPSDB ASCII implementation, packet identifiers are derived from the PID_DESCR field if the packet is an instrument specific packet, otherwise the identifier is constructed from the PID_APID, PID_TYPE, PID_STYPE, PID_PT1_VAL and PID_PT2_VAL fields.

For the HPSDB PAL implementation, only HIFI packet identifiers are derived from the PID_DESCR fields, all other packet identifiers are constructed from the PID_APID, PID_TYPE, PID_STYPE, PID_PT1_VAL and PID_PT2_VAL fields.

		MIB ASCII	MIB PAL	HPSDB ASCII	HPSDB PAL
Parameter Id	**HIFI**	PCF_DESCR	PCF_DESCR	PCF_DESCR	PCF_DESCR
	PACS	PCF_DESCR	PCF_DESCR	PCF_DESCR	PCF_NAME
	SPIRE	PCF_DESCR	PCF_DESCR	PCF_DESCR	PCF_NAME
	SPACECRAFT	PCF_DESCR	PCF_DESCR	PCF_NAME	PCF_NAME
Packet Id	**HIFI**	PID_DESCR	PID_DESCR	PID_DESCR	PID_DESCR
	PACS	PID_DESCR	PID_DESCR	PID_DESCR	APID_TYPE_STYPE_PID1_PID2
	SPIRE	PID_DESCR	PID_DESCR	PID_DESCR	APID_TYPE_STYPE_PID1_PID2
	SPACECRAFT	PID_DESCR	PID_DESCR	APID_TYPE_STYPE_PID1_PID2	APID_TYPE_STYPE_PID1_PID2

Table 1: Describes for each implementation base how the identifiers for parameters and packets are derived.

[6] An instrument parameter name starts with H, P, or S for HIFI, PACS or SPIRE respectively.

6.6. The tm-file Data Format

The Telemetry file (tm-file) is a binary file containing all telemetry packets from a particular observation. The format is very simple as it just contains blocks of information with the following format.

long = FineTime	int = length	byte[length] = Packet

Each block contains,

1. in the first 8 bytes, the time at which the packet was written to the file
2. in the next 4 bytes, the length of the packet, and
3. in the next 'length' bytes, the packet itself.

Telemetry files can contain both telemetry and tele-command packets, but the latter are skipped during the import process. Use the PacketReader to read a tm-file into a PacketSequence as follows:

```
seq = PacketSequence(PacketReader("/data/pacs/FM_FLIGHT/OD0400-0499/OD0464.tm"))
```

Appendix A: User Commands

The PacketSequence

Description

This HIPE command loads a packet sequence into your session.

Synopsis

```
seq = PacketSequence([Reader])
```

Arguments

PacketReader

What is a packet reader?

Examples

```
seq = PacketSequence(PacketReader("/Users/rik/data/pacs/OD0420.tm"))
```

Appendix B: Archived JIRA Issues

HCSS-5930 • PacketReader iterator has implementation flaw

This issue is not at all serious and we didn't see any processing failures due to this issue. The description and analysis of the issue is here for reference.

Description

The way the PacketReader iterator is implemented currently has a drawback for developers who like to use this iterator in a Java foreach construct or while loop.

The hasNext() method actually already constructs the next packet and can therefore throw a number of unexpected RuntimeExceptions. With a foreach or while loop these exceptions can only be catched if a try..catch is placed around the loop resulting in the complete iteration to abort in case of a corrupt packet.

Analysis

The hasNext() method currently catches an IOException and than returns false to end the loop, which is as expected. The hasNext() method however should indeed not try to construct the next packet because this can result in a RuntimeException which aborts the iteration as described above. Instead, the construction of the packet must be delayed until the next() method is called. hasNext() must be restricted to read in the data stream and throwing IOExceptions if this fails.

Glossary

Framework

A framework is a collection of classes, tools, and APIs to help the different components in a system to work together. Find a good definition of a software framework at Wikipedia.

packet

A packet is a bunch of parameter values that is downlinked from the satellite. A packet is an array of bytes with a header that allows to decode the information contained. A packet has a fixed definition that is described in a mission database.

parameter

A measurable entity of an instrument or the satellite. Sampled values of all parameters will be downlinked in telemetry packets.

tm-file

A shortcut for telemetry file. Usually the file is expected to have the extension `.tm`, but this is not imposed by the binstruct library.

telemetry file

A binary file that contains telemetry packets.

services

An interface for reading telemetry definitions from PIB, MIB, or HPSDB. This is easily extendable to other formats like e.g. XML.